UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÀTICA DE BARCELONA

Master in Innovation and Research in Informatics

MASTER THESIS

A configurable geometry processing system

Joan Fons Sanchez

October, 2019

Directors:

Carlos Andujar, Antonio Chica, Nuria Pelechano

Contents

Introduction

Polygonal meshes have become the most used 3D surface representation model over the last years. There are plenty of tools dedicated to the creation and manipulation of such models, from photogrammetry scans to completely artist authored meshes. As in many other fields, this rich ecosystem of models and tools raises the need of task automation. Scans have to be cleaned and simplified, some applications need procedurally generated meshes and in many cases, artist authored meshes require some pipeline processing to make them suitable for their final application.

While some of the existing tools provide different kinds of scripting or automation functionalities, we have not found any one of them that focused on automation itself. The main goal of *Meshpipe* , the tool developed during this master's thesis, is to ease the process of creating scripted pipelines by providing a simple python API and a basic visual environment that allow for fast iteration and quick visualization of the pipeline results.

1. Motivation

This project has been developed in the context of a research group mostly focused on graphics, virtual reality and data visualization as a whole. This means that, over the years, many situations required working with 3D polygonal meshes.

As in any engineering problem, the general approach is to always look for existing tools that fit the job and, if none are found, produce some of your own. Given the fact that research is intrinsically innovative, most of the time there is no fitting tool for the job, but some parts of the problem can indeed be handled by already existing software.

The aim of *Meshpipe* is to provide a base ground, a collection of basic tools with a simple python API that will allow the user to quickly put together a mesh processing pipeline. The framework is designed with extensibility in mind and also provides a viewer application that helps visualize the pipeline results as well as any intermediate states along the pipeline.

2. State of the art

2.1 Mesh processing

Mesh processing or geometry processing is the research field in which applied mathematics and computer science meet to produce algorithms and techniques that involve 3D polygonal meshes.

A polygonal mesh is a model used for 3D surface representation. It consists of a set of points in 3D space called vertices, a set of connections between vertex pairs called edges and a set of closed edge loops called faces. This representation is very flexible and allows for modeling virtually any imaginable 3D surface but it has less precision than other surface representations based on 3D curves, especially when the surface to represent is a continuously smooth curve. However, this precision problems can be mitigated by increasing the amount of vertices and faces. Since the amount of processing power has increased a lot over the last decade the balance has tipped over the fact that polygon meshes are much more flexible than other representations and are the most used model when dealing with 3D surfaces.

2.1.1 The mesh life cycle

The processing of a mesh involves three main stages and these stages are form the life cycle of a mesh. First, the mesh is ”born” or created by one of these three methods: an artist authored model, a procedural generation, or a scan.

Once we have a starting mesh, it can be analyzed and edited repeatedly in a cycle. This process usually involves getting different measurements, such as the distances between vertices, its smoothness or its Euler characteristic. Editing the mesh may consist in smoothing, removing certain elements, deforming or performing rigid

transformations, among many others.

The final stage in the ”life” of a mesh arrives when it is consumed. This can mean it is consumed by a viewer as a rendered asset in a game or movie, or fabricated in the real world through some computerized production machine such as a 3D printer. The end use of a mesh can also be a simple decision about it, like whether or not it satisfies some criteria.

2.2 Existing tools

There are two major families of mesh processing tools: the fully visual editing software and the toolkit libraries. The first ones offer a graphical interface and a set of features and operations that can be performed over a mesh that has been previously loaded by the user.

The second group consists of software libraries. These are meant to be used in custom processing pipelines and provide the data structures and basic algorithms to produce more complex mesh modifications or analysis.

2.2.1 Software

Meshmixer

Meshmixer is a closed source software owned by *Autodesk*. Its main focus is to provide tools for mesh processing and ”clay-like” sculpting. It has a very powerful set of tools and a 3D viewport that helps the user visualize the mesh properties as well as perform all sorts of modification tasks (see figure 2.1).

Over the last year it has incorporated many features related to 3D printing, such as support generation for manufacturing processes that can not handle overhangs in the mesh geometry or integration with various 3D printer models in order to

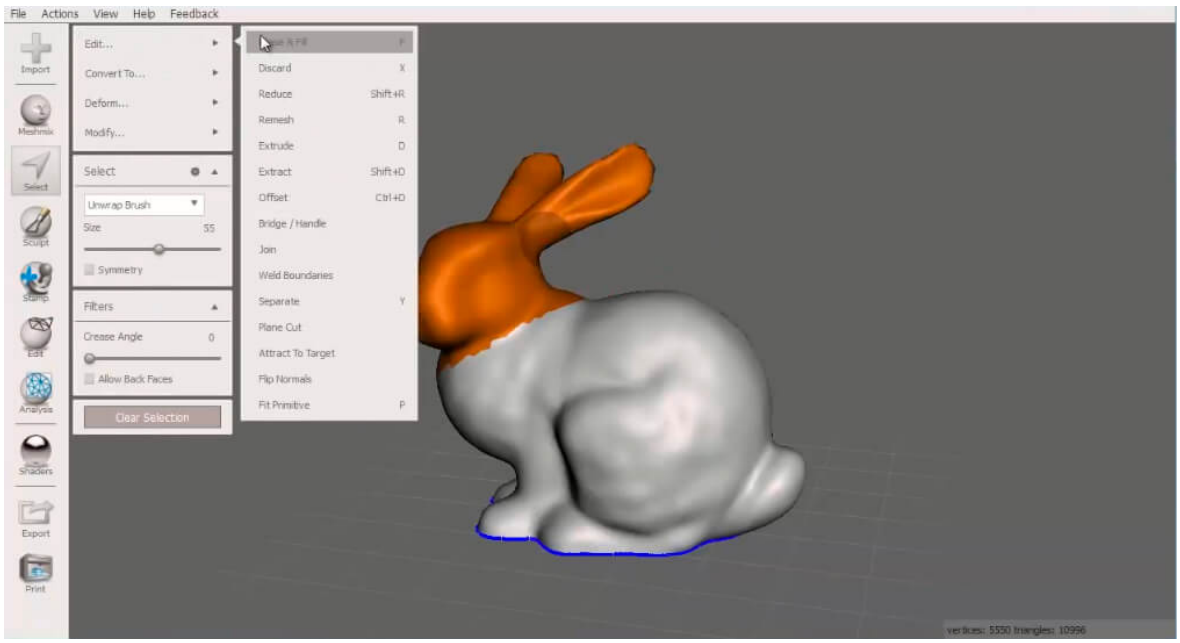

Figure 2.1: Screenshot of a user performing a selection operation in Meshmixer.

provide a one-click setup to start the prints.

However, there is no support for any kind of automation or scripting.This means that every mesh has to be processed individually and every operation has to be performed manually by a user.

MeshLab

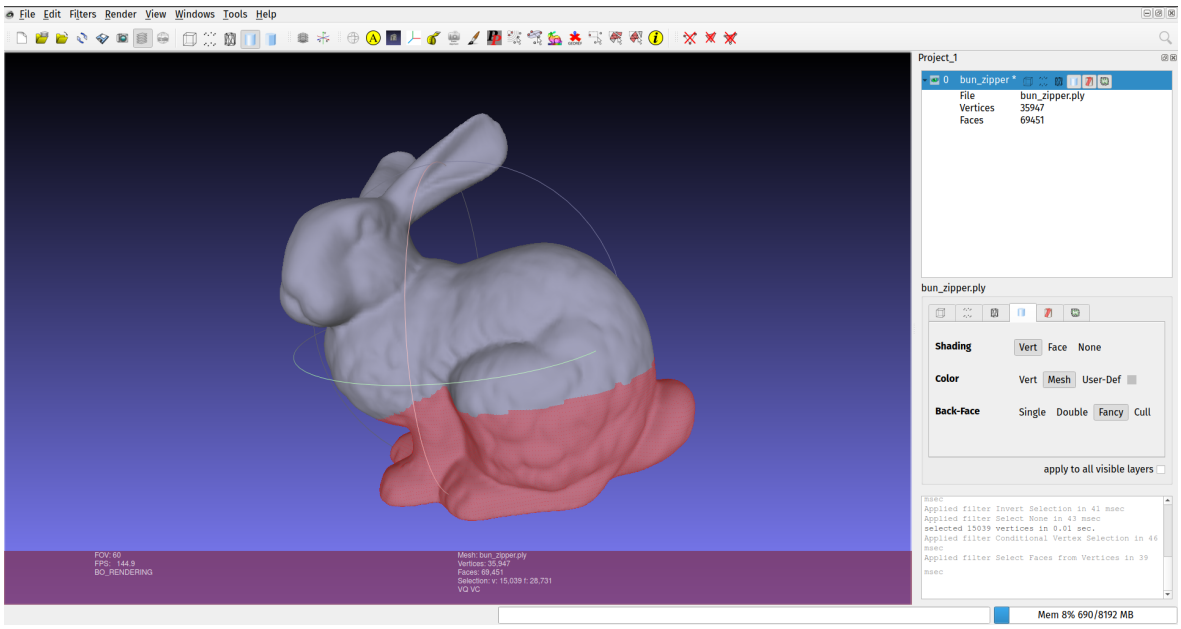

Figure 2.2: MeshLab viewer with a partially selected model.

Meshlab [1] is a free and open source tool for mesh processing that excels in the acquisition and reconstruction of meshes (see figure 2.2). It has great tools for working with 3D scanned data and for obtaining a clean and usable mesh out of it. Additionally, there is a web based variant called *MeshLabJS*, that replicates many of the features found in the original software in a web browser environment, thus requiring no local installation.

Unlike *Meshmixer*, this tool introduces the concept of "filter scripts" which allow the user to define a processing pipeline to be reused in as many meshes as needed. However, these pipelines are fixed, they only contain a list of predefined steps and parameters and there is no possibility of changing them depending on the properties of the mesh. On top of that, the filter script files use a rather convoluted format that produces many incompatibilities between *MeshLab* versions and makes it hard to generate custom filter scripts using other scripting languages.

OpenFlipper

OpenFlipper [2] is a very powerful mesh processing suite (see figure 2.3). It is developed under a permissive open source license and it focuses on extensibility. It provides all the basic elements of a mesh processing software: viewer, selection and basic tools and data structures (based on *OpenMesh*, see section 2.2.2), while exposing a complete plugin development API.

This plugin system can load both C++ and Python plugins and can be used to add new UI elements to the software and implement virtually any possible new feature. However, this flexibility comes at a cost, since many tools are created and manitained by different developers. There is not a clear common interface for tools and, for example, not every interactive action has a one to one mapping with an API call, making it hard to automate some tasks.

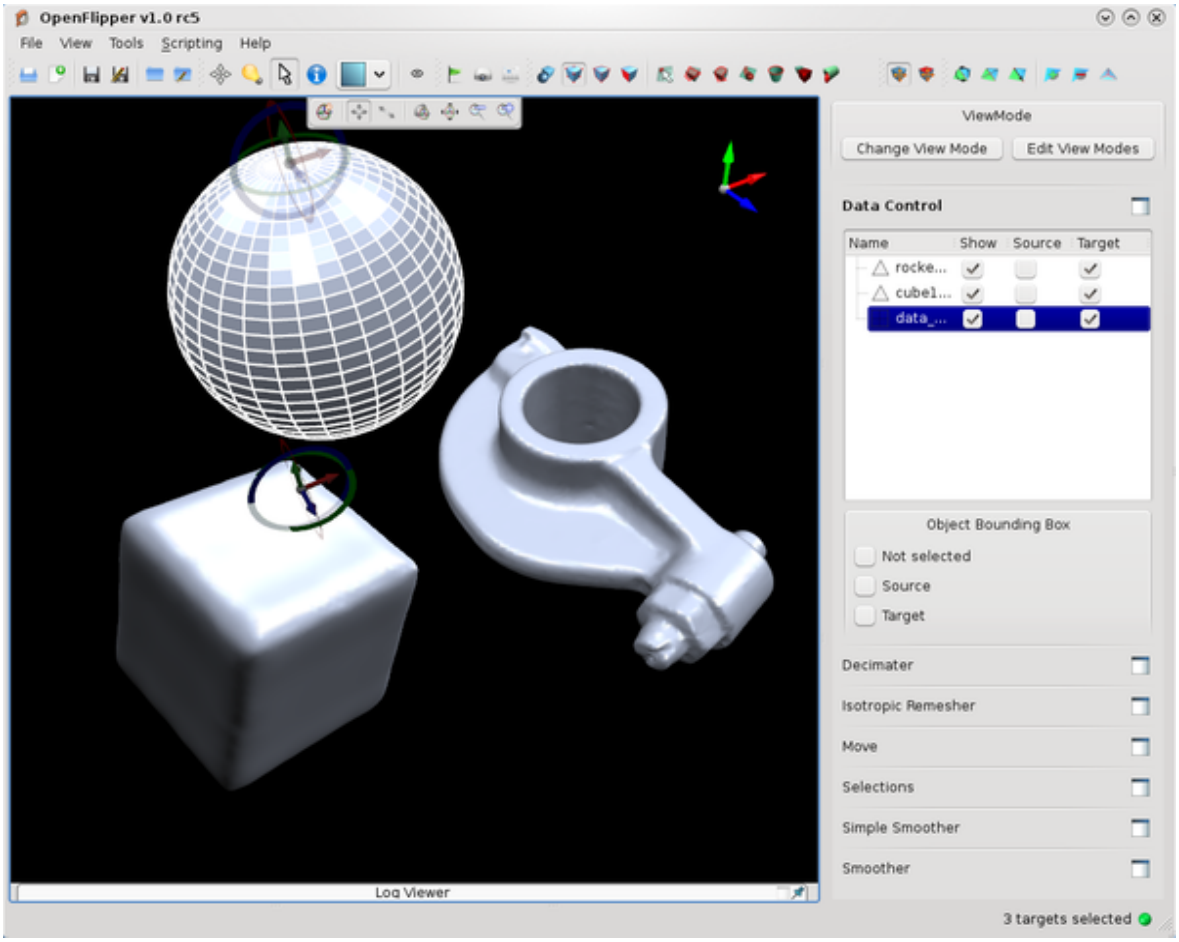

Figure 2.3: Basic interface of *OpenFlipper*.

2.2.2 Toolkits

There are many software libraries that provide the basic ground for working with polygonal meshes. Given that mesh processing usually requires good performance almost all of them are written in C++. The following is a brief explanation of some of the most widely used mesh processing libraries.

OpenMesh

This free and open source library is developed by the *Visual Computing Institute* in the RWTH Aachen University. It provides the bare basics for developing a mesh processing algorithm and recent versions also include implementations for mesh decimation, smoothing and subdivision.

It implements a half-edge based data structure [3] for storing mesh information as well as some infrastructure for storing mesh properties i.e. store some data for every element in the mesh. This makes it suitable for any job that requires fast queries for traversing the mesh and/or generating new geometry but it does not include any of the more advanced mesh processing algorithms.

CGAL

The *Computational Geometry Algorithms Library* (CGAL) is by far the most extensive open library in the context of computational geometry. The project is developed via a collaborative effort from a community of developers working at research institutes, universities and companies and it includes over a hundred modules, all providing some algorithm or core functionality to this huge ecosystem.

It is important to keep in mind that mesh processing is only a part of the whole computational geometry scope, so only a subset of the CGAL modules will be actually useful when developing a mesh processing program. Still, CGAL has the highest amount of geometry processing [4] techniques and algorithms implemented out of the box. Some of this algorithms include mesh subdivision [5], simplification [6], deformation [7] and parametrization [8].

PMP

The *Polygon Mesh Processing* library (PMP) [9] is an MIT-licensed mesh processing toolkit. It has its origins on the *OpenMesh* half-edge implementation but evolved a lot form it. It provides all the usual tools of any modern mesh processing library: a core mesh data structure plus a set of well known algorithms to work with it; but it stands out from the rest because of its clean and simple integrated 3D viewer.

This example implementation of a 3D viewer can display meshes straight from the library's *Mesh* data structure without any extra hassle for the developer. Coupled with the fact that the user interface is easily extendable, makes the PMP viewer

a boilerplate for developing demos and mesh processing applications that require some kind of user interaction. For that reason we decided use PMP as the core library for *Meshpipe* , extending the viewer to suit our needs and adding any extra algorithms and features where we needed them.

3. Tool design

In this section we will elaborate on the different design decisions that were made at the start of the project, giving an overview of the ideal goal that we are trying to achieve. These design goals have been kept as much as possible during the development phase, but some parts have not been implemented due to time constraints.

As a general overview, our goal is to develop a clean and simple C++ API with a Python counterpart that matches it as closely as possible. This API will be focused on providing the tools to build fully automated mesh processing pipelines, thus it will include various selection tools and operators and a set of modifying operators that can be applied to a given selection.

On top of that API, we want to develop a 3D viewer that allows to quickly test and visualize the result of various API calls as well as provide a set of interactions for the user to select parts of the mesh or apply transformations using the mouse. The key component of these user interactions is that every interactive operation has an equivalent API call that will appear on the application's console once applied. This will allow the user to save that API call and use it in an automated pipeline without having to manually perform the same operation again and again.

3.1 API design

3.1.1 Overview

The *Meshpipe* API has three major components: the *Mesh* class, the *Selection* class, and a set of extension modules that are built on top of the other two core classes. The *Mesh* and *Selection* core classes work together to allow the user to traverse a mesh and select the key elements in it while the various extension modules provide

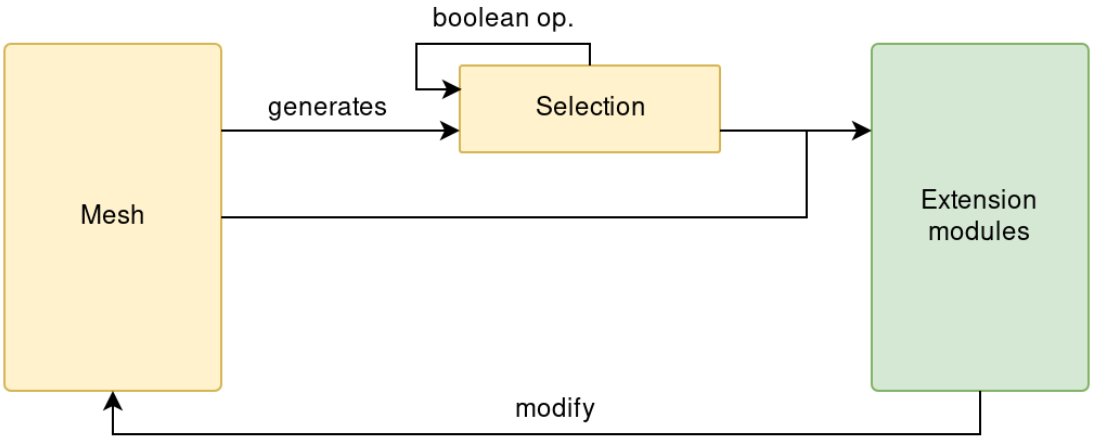

Figure 3.1: Property transfer diagram. M is the simplified mesh, S is the original mesh, r is the intersection line and P is the final sampled point.

different modifying operations that can be applied to a given mesh and selection (see figure 3.1).

We decided to use this global organization because we want to keep a clean and simple core library with only the essential tools while at the same time make it easy to extend the library to add new algorithms or mesh processing techniques.

3.1.2 The Mesh class

The *Mesh* class is the essential core of the *Meshpipe* library. It is mainly based on PMP's *SurfaceMesh* class (see section 2.2.2) which is an implementation of the classic half-edge data structure. This data structure allows for quick traversal of the mesh while keeping the memory requirements relatively low.

Half-edge representation of meshes stores all the topological relations on the edges. Each edge references its two vertices, the faces it belongs to and the two next edges in these faces. Edges are split (i.e. an edge connecting vertex A and vertex B becomes two directed half-edges from A to B and vice versa). Figure 3.2 illustrates the way connectivity is stored in this structure, which can be listed as:

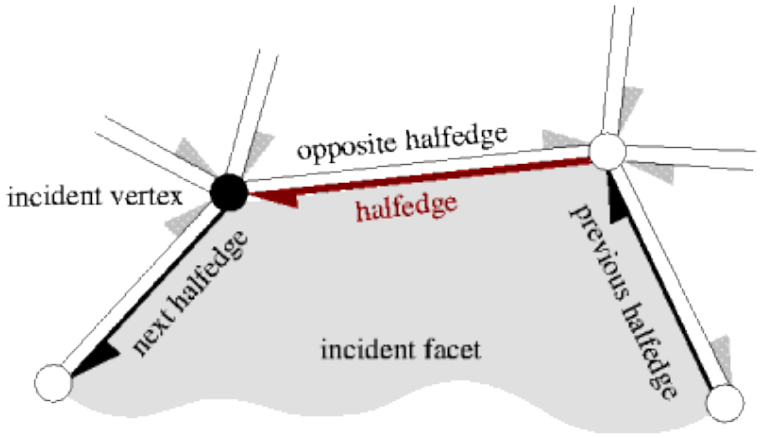

Figure 3.2: Half-edge connectivity diagram.

- Each vertex references one outgoing halfedge, i.e. a halfedge that starts at this vertex.
- Each face references one of the half-edges bounding it.
- Each halfedge provides a handle to:
  - the vertex it points to.
  - the face it belongs to.
  - the next halfedge inside the face.
  - the opposite halfedge.
  - optionally, the previous halfedge in the face.

With these links between items, we can circulate around the mesh and find all the other topological relations we may need. For example, if we want to iterate over all vertices in a face, we can just start at its referenced halfedge and keep moving to the next half-edge until we reach the starting one.

Mesh traversal

As you can see in listing 1, the basic element of the mesh traversal is the halfedge as it allows to navigate to other halfedges as well as other elements in the mesh. The rest of navigation methods are just helper methods that do the necessary jumps and conversions to navigate between different elements and neighborhood types.

```
# Get a halfedge that starts at the given vertex
halfedge(vertex)
# Get the next halfedge within the incident face
prev_halfedge(halfedge)
# Get the previous halfedge within the incident face
next_halfedge(halfedge)
# Get the opposite halfedge in the same edge
opposite_halfedge(halfedge)

to_vertex(halfedfe) # Get vertex at halfegde origin
from_vertex(halfedge)# Get vertex at halfegde destination
vertices(face)       # Get iterator over all vertices in a face
vertices(vertex)     # Get iterator over adjacent vertices
vertex(edge, i)      # Get vertex from edge
# ...

halfedge(edge, i)# Get a halfedge form an edge
halfedge(face)   # Get a hafledge from the given face
# ...

edge(halfedge)   # Get edge from a halfedge

faces(vertex)    # Get all faces adjacent to a vertex
face(edge, i)    # Get face adjacent to an edge
# ...
```

Listing 1: Some (not all) of the mesh traversal methods.

Selection creation

The *Mesh* class is also in charge of generating the most basic selections, see listing 2. These are meant to be the basic building blocks which can be later combined or modified using the methods in the *Selection* class.

As an example, a call to the `select_boundary_vertices()` method generates a selection including all the vertices that lay in the mesh boundary. This selection can be expanded using the `expand()` method in Selection or combined with other selections using the provided boolean operations.

Another interesting method in this family is the `select_vertices_by_expr()`. This is a really powerful operation since it takes as a parameter an expression, that gives users full power to select vertices by whatever criteria they want to implement.

Finally, the *Mesh* class contains a set of user-interaction related methods. These al-

low for reproducibility of user actions independently of the mesh, see section 3.2.1 for more details.

```
empty_vertex_selection()
# halfedge, edge, face...

select_all_vertices()
# halfedges, edges, faces...

select_boundary_vertices()
select_non_manifold_vertices()
# halfedges, edges, faces...

select_self_intersecting_edges()
select_self_intersecting_faces()

select_vertices_by_expr(expression)
# halfedges, edges, faces...

select_random_vertices(prob)
select_random_vertices(prob_expression)
# halfedges, edges, faces...

select_vertices_inside(volume)
# halfedges, edges, faces...

lasso_select_vertices(camera, lasso_points)
click_select_vertices(camera, point)
# halfedges, edges, faces...
```

Listing 2: Selection creation methods. There are similar versions of these methods for every topological element: vertices, halfedges, edges and faces.

Selection conversion

Another task of the *Mesh* class is to convert between selection types. A user may want to select a set of vertices and after that, select all the edges that are connected to at least one of the selected vertices. The methods in listing 3 are the ones in charge of this kind of conversion, and they have some parameters that slightly change the conversion behavior. For example, when converting from a vertex selection to a face selection a minimum number of selected vertices can be specified so that only faces with 2 (or however many) vertices in the original selection get selected.

```
vertex_to_edge_selection(sel, both)        # only select edges with both
                                           #   vertices selected
vertex_to_halfedge_selection(sel)
vertex_to_face_selection(sel, num_verts)   # minimum number of vertices

halfedge_to_vertex_selection(sel)
halfedge_to_edge_selection(sel)
halfedge_to_face_selection(sel, num_he)    # minimum number of halfedges

edge_to_vertex_selection(sel, num_edges)   # minimum number of edges
edge_to_halfedge_selection(sel)
edge_to_face_selection(sel, num_edges)     # minimum number of edges

face_to_vertex_selection(sel, num_faces)   # minimum number of faces
face_to_edge_selection(sel, both)          # only select edges with both
                                           #   adjacent faces selected
face_to_halfedge_selection(sel)
```

Listing 3: Selection conversion methods.

Properties

Properties or attributes are a very important element in the world of mesh processing. They allow for storage of some data per element in the mesh. This data can be of many different types and come from different sources. As an example, we can store color information for every vertex in a mesh that has been reconstructed using photogrammetry; or we can store a floating point value for every edge in the mesh, representing the maximum stress level that any given edge has recorded during a mechanical simulation.

It is important that this kind of data is easily accessible both for reading and writing and that we only store in memory the property types we actually use. So we have decided to go with a separate *Property* class approach. Adding a new property to a mesh returns a Property type object that can be accessed as an indexed container and each element can hold a value of one of the following types: int, float, Vec2, Vec3 and Vec4.

Listing 4 shows some of the basic methods for adding, retrieving and removing properties from a mesh, and listing 5 displays a little usage example.

```
# Float property per vertex
add_vertex_float_property(name, default_value)
get_vertex_float_property(name)
remove_vertex_float_property(property)
has_vertex_float_property(name)
vertex_float_property(name, def_val)    # Gets property if exists,
                                        # creates a new one otherwise

# Vec3 property per face
add_face_vec3_property(name, default_value)
get_face_vec3_property(name)
remove_face_vec3_property(property)
has_face_vec3_property(name)
face_vec3_property(name, def_val)       # Gets property if exists,
                                        # creates a new one otherwise

# Equivalent methods exist for every combination of data type and element
```

Listing 4: Property management methods.

```
# Allocate a property storing a Vec3 per edge
prop = mesh.add_edge_vec3_property("my_property")

# Access the edge property like an array
prop[e] = Vec3(1.0, 1.0, 1.0)
print(prop[e])          # prints <1.0, 1.0, 1.0>

# Remove property and free memory
mesh.remove_edge_vec3_property(prop)
```

Listing 5: Simple example of properties usage in Python.

Mesh generation and manipulation

In order to generate a mesh from scratch only two operations are required: the addition of vertices and the addition of faces. With the addition of a couple of convenience methods the resulting API can be found in the listing 6, as well as a simple example.

```
# Available methods
add_vertex(position)
add_triangle(v0, v1, v2)
add_quad(v0, v1, v2, v3)
add_face(vertices)

# Example
mesh = Mesh()

v0 = mesh.add_vertex(Vec3(0.0, 0.0, 0.0))
v1 = mesh.add_vertex(Vec3(1.0, 0.0, 1.0))
v2 = mesh.add_vertex(Vec3(0.0, 1.0, 0.0))
v3 = mesh.add_vertex(Vec3(1.0, 1.0, 1.0))

mesh.add_triangle(v0, v1, v2)
mesh.add_triangle(v0, v1, v3)
```

Listing 6: Mesh generation methods with example.

Miscellaneous

The sections above have highlighted the most important parts of the *Mesh* class inside the *Meshpipe* API. However, there are many more utility methods that do not belong in any special category. Getting the total number of vertices, getting the position of a vertex and checking whether a vertex belongs to the mesh boundary are just some of the many other methods that help the user write a properly working pipeline script.

3.1.3 The Selection class

The *Mesh* class holds all the mesh data and allows for some low level modifications, but the bulk of the mesh transformations are done via operators and modifiers. These operators can often be applied to the whole mesh or just to some parts of it. For example, we may want to smooth only some part of the mesh or completely delete some elements while keeping the rest of the mesh intact.

In order to specify which elements will be modified and which not, we introduced the *Selection* class. There are four different types of selection, all inheriting from the

same *Selection* class: *VertexSelection*, *HalfedgeSelection*, *EdgeSelection* and *FaceSelection*. Each one of the classes holds a list of all the selected and unselected elements of its type and provides a way to iterate the selected elements. On top of that the *Selection* class holds some methods to select or deselect elements, a group of topology-based operations (i.e. grow or flood the selection), and a set of boolean operators to generate more complex selections; all these methods can be found in listing 7.

```
selected()            # Selected elements iterator
n_selected()          # Get selection size
is_selected(element)  # Check if selected


# Select/deselect single element
select(element)
deselect(element)

# Select/deselect list of elements
select(elements)
deselect(elements)

# Topology operators
grow(mesh, steps)
shrink(mesh, steps)
flood(mesh)

# Boolean operators
invert()
subtract(selection)
combine(selection)
intersect(selection)
```

Listing 7: All *Selection* methods.

This reduced number of tools, together with the selection creation methods in the *Mesh* class, turn into a very powerful combination. Listing 8 shows a typical use case where we want to detect small floating pieces of mesh that are generated during a scan and get rid of them.

```
delete_selection = mesh.empty_vertex_selection()

for v in mesh.vertices():
    if delete_selection.is_selected(v): continue

    s = mesh.empty_vertex_selection()
    s.select(v)
    # Flood selection to all vertices in the same connected component
    s.flood()
    if s.n_selected() < 10:
        delete_selection.combine(s)

mesh.delete(delete_selection)
```

Listing 8: Example of selection and deletion of small connected components.

3.1.4 Extension modules

During the early development of *Meshpipe* , all mesh operators were encapsulated inside the *Mesh* class. Simplification, normal calculation and property transfer were all sharing the same space inside the class, and it was growing bigger and bigger with every newly added feature.

We quickly realized that, if we wanted our API to be expandable, we needed a better solution. That is why we moved every non essential operator out of the mesh class and separated them into their own modules. This separation makes it possible to keep expanding the library features without ending up with a huge and unmaintainable *Mesh* class and, at the same time, reduces clutter in the user facing API: user scripts can choose to import only the modules they need instead of having all the functionality compacted in a single class.

Simplification

Mesh decimation is the process of reducing the amount of geometry in a mesh while trying to keep the overall original shape as much as possible. It is one of the basic tools available in most mesh processing toolkits and in *Meshpipe* it has its own module.

The simplification module currently only houses one method, an edge-collapse mesh simplification, but could be expanded to implement other simplification algorithms in the future.

As can be seen in listing 9, the edge collapse method can take various parameters that determine the complexity of the simplified mesh as well as its quality. The user can define limits in different metrics to ensure the resulting mesh will not deviate too much from the original as to make it unusable in its final application.

```
from meshpipe import MeshSimplification

MeshSimplification.simplify(mesh, selection, n_vertices, aspect_ratio,
    edge_length, max_valence, normal_deviation, hausdorff_error)
```

Listing 9: Mesh simplification module.

Analysis

Another important aspect of mesh processing is the various analytic measures that can be performed over any given mesh. *Meshpipe* provides only some basic measurements but again, this could be expanded in future versions.

The current implementation supports various curvature metrics (minimum, maximum, mean and gaussian) as well as per-vertex, per-face and per-halfedge normal computation. All these method definitions can be found in listing 10.

```
from meshpipe import MeshAnalysis

# Curvature results are stored in a per-vertex property
prop = MeshAnalysis.compute_min_curvature(mesh)
prop = MeshAnalysis.compute_max_curvature(mesh)
prop = MeshAnalysis.compute_mean_curvature(mesh)
prop = MeshAnalysis.compute_gaussian_curvature(mesh)

# Normals are also stored as a property
prop = MeshAnalysis.compute_vertex_normals(mesh)
prop = MeshAnalysis.compute_face_normals(mesh)
prop = MeshAnalysis.compute_halfedge_normals(mesh, crease_angle)
```

Listing 10: Mesh analysis module.

Parametrization

In the field of mesh processing, parametrization is the process of defining a new coordinate system that spans over the surface of the mesh, giving each point on the surface a corresponding set of coordinates. This parametrization is mostly used to generate or apply texture data on top of the mesh surface, greatly increasing the amount of surface information without actually increasing the amount of topology elements.

Since most of the time parametrization is used to work with 2D textures, our library (as many others) only offers a 2D parametrization algorithm. Even more so, it exposes a texel density parameter so the user can specify how many texels are going to fit in a unit-length line across the mesh surface. Given the nature of the parametrization process, this density parameter can not be kept as an exact constant across the whole surface, but the algorithm makes a best effort: it compute a final texture size that results in a number of texels per unit that is as close as possible to the requested one.

The parametrization process usually has two requirements: injectivity and minimization of distortion. Although some applications make use of it, non-injectivity is often an annoyance, especially in automatically generated parametrizations. Having a non-injective parametrization means that two separate points on the surface share the same coordinates, making it hard to work with texture data: one texel stores values for multiple surface spots which very likely do not share the same properties.

Another very important metric of any given 2D parametrization is the curvature distortion. By stretching a curved 3D surface on top of a plane the relative distances between a point on the mesh surface and the relative distances between their parametric counterparts get distorted. This phenomenon is inevitable but efforts can be made to keep the distortion as low as possible. The type and amount of distortion is basically intrinsic to each parametrization algorithm, but in our case we use a variation of the LSCM [10] which tries to reduce distortion by minimizing angle deformations and non-uniform scalings.

The specific API method to generate a 2D parametrization for a mesh can be found in listing 11.

```
from meshpipe import MeshParametrization

# Parametrization is stored in a per-halfedge property named "h:tex"
texture_size = MeshParametrization.parametrize(mesh, texels_per_unit)
```

Listing 11: Mesh parametrization module.

Property transfer

As explained in the previous section, 2D data textures are often used to store surface information without having to actually represent it in the mesh topology. A very typical use case is to generate a simplified version of a mesh and transfer the geometry detail from the original mesh to the simplified one.

We used a simple yet effective method for transferring properties between meshes, first proposed by Cignoni et al. [11]. In order to compute the appropriate value for each texel in the transfer texture we first need to find its corresponding position and surface normal on the simplified mesh. We get this information by ”plotting” all the triangular faces into two temporary textures: one containing the positions and the other containing the surface normals. All the values in these textures are interpolated using the texel's barycentric coordinates to obtain a set of smooth an continuous values across each face surface.

Once we know the position T and surface normal N of a texel, all we need to do is find the point P of intersection between the line r parallel to N that goes through T, and the full detail mesh. The value of the texel we are computing will be the same as the value of the intersection point on the original mesh. This value on the original mesh can be automatically computed by interpolating a mesh property or by a user defined lambda function. By defining a custom lambda function users not only gets more control over the interpolation method, they can also define properties that have different values at a texel level; something which is not achievable with the property system since values are always attached to a mesh element rather than a

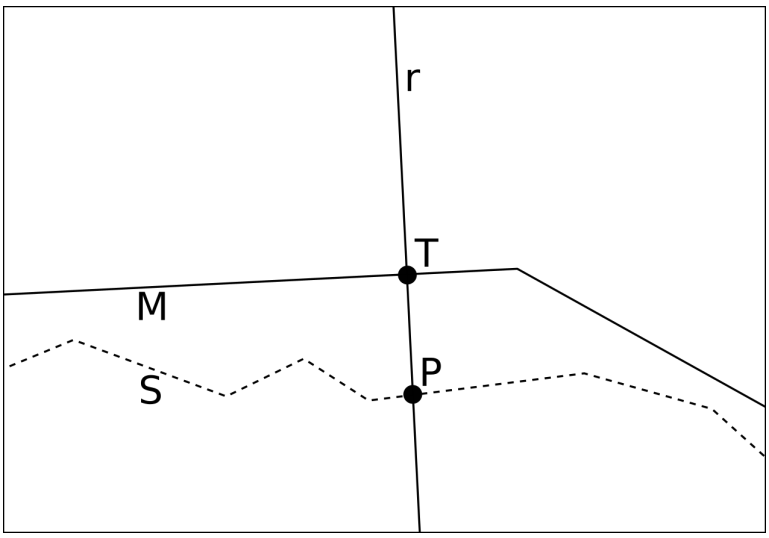

Figure 3.3: Property transfer diagram. M is the simplified mesh, S is the original mesh, r is the intersection line and P is the final sampled point.

surface point. You can see the definitions of both property transfer options in listing 12.

```
from meshpipe import MeshPropertyTransfer as mpt

# Default property transfer, similar set of methods for every basic type
mpt.transfer_vertex_float_property_to_texture(source_mesh, target_mesh,
    prop, result_texture_path, value_range = Vec2(0, 1))
mpt.transfer_halfedge_float_property_to_texture(source_mesh, target_mesh,
    prop, result_texture_path, value_range = Vec2(0, 1))
mpt.transfer_face_float_property_to_texture(source_mesh, target_mesh,
    prop, result_texture_path, value_range = Vec2(0, 1))

mpt.transfer_vertex_vec3_property_to_texture(source_mesh, target_mesh,
    prop, result_texture_path, value_range = Vec2(0, 1))
mpt.transfer_halfedge_vec3_property_to_texture(source_mesh, target_mesh,
    prop, result_texture_path, value_range = Vec2(0, 1))
mpt.transfer_face_vec3_property_to_texture(source_mesh, target_mesh,
    prop, result_texture_path, value_range = Vec2(0, 1))
# ...

# Custom property definition, similar method for every basic type
mpt.transfer_custom_float_property_to_texture(source_mesh, target_mesh,
    prop_lambda, result_texture_path, value_range = Vec2(0, 1))
mpt.transfer_custom_vec3_property_to_texture(source_mesh, target_mesh,
    prop_lambda, result_texture_path, value_range = Vec2(0, 1))
# ...

# Example call for a custom property definition
mpt.transfer_custom_float_property_to_texture(source, target,
    lambda b, he, p: distance(p,Vec3(1,2,3)), "distance_tex", Vec2(0,3))
```

Listing 12: Property transfer module methods.

Point clouds

In many mesh processing algorithms, being able to work with a point cloud data set is crucial. Since a point cloud is just a set of points without any topological relationship, the two basic operations that can be performed on them are: finding all the points within a radius and finding the K nearest neighbors.

Point cloud analysis is specially useful when working on surface reconstruction from a set of scanned points, but its uses are not limited to only that. Some mesh processing pipelines can take advantage of finding close vertices in an already topologically rich mesh; for that reason we decided to not differentiate a point cloud from a regular *Mesh* from the API's perspective.

The point cloud module takes a *Mesh* object as input and, if users desire to work with pure point cloud data, they can just build a *Mesh* with only vertices and no other topological elements. This makes the API suitable for a wide variety of usages without having to take extra conversion steps. Listing 13 shows a little usage example of the point proximity methods in this module.

```
from meshpipe import MeshPointCloud

MeshPointCloud.find_vertices_in_sphere(mesh, point, squared_radius)
MeshPointCloud.find_k_nearest_vertices(mesh, point, k)

# Example, prints the 5 nearest vertices to the (1,2,3) point
neighbors = MeshPointCloud.find_k_nearest_vertices(mesh, Vec3(1,2,3), 5)

for neighbor in neighbors:
    p = mesh.position(neighbor)
    print(p)
```

Listing 13: Point cloud module methods and usage example.

3.2 Viewer design

The other main element in the *Meshpipe* ecosystem is the 3D viewer (see figure 3.4. Its main purpose is to provide a quick testing iteration environment as well as some user interactive tools (i.e. lasso selection, mouse based deformations, etc.).

3.2.1 Quick testing iteration

When developing a mesh processing pipeline, usually at some point one parameter needs to be tweaked or some changes need to be made iteratively until a good solution is found. In these cases testing iteration time is very important.

Testing iteration time can be defined as the amount of time that passes between making a change and being able to verify the effects of said change. From our experience in developing mesh processing pipelines using C++ toolkits, this can add up to quite a lot of time.

Compiling processing pipeline, waiting for the mesh to be processed, switching to a 3D editing software and loading the pipeline's result; these are the typical steps to go through before being able to inspect the final result. This slows down the development process considerably, so we wanted to minimize iteration times as much as possible.

Input data

Very often, when developing a processing pipeline, only a subset of the real input data is used. For example, the developer may want to focus on a specific part of the input mesh or just wants to use a simplified version of it in order to reduce processing times.

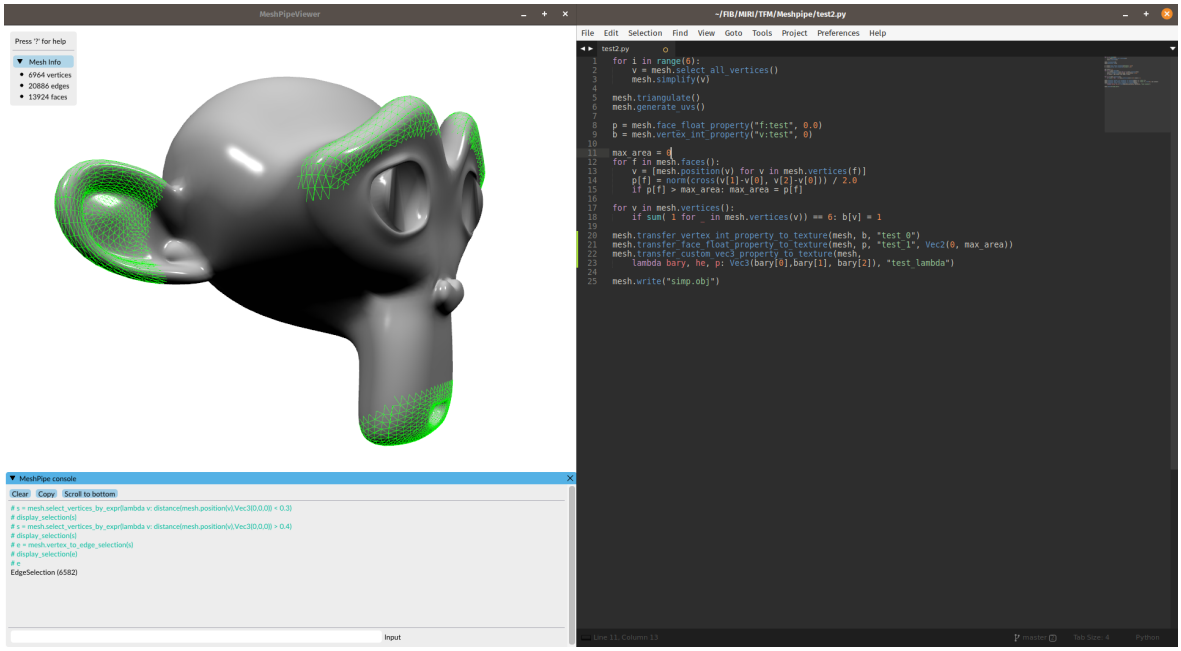

Figure 3.4: Intended usage of the *Meshpipe* viewer: side by side with a text editor.

Once the pipeline is setup and working for the reduced input data, the developer proceeds to test the pipeline on the real data set, maybe going through multiple steps of increasing data size. *Meshpipe* tries to help this whole process as much as possible.

Every mouse driven operation that can be performed in the 3D viewer has its own translation on the Python API. This API calls are displayed in the 3D viewer console every time the users perform an operation, so they have the option to copy the Python code and paste it directly on their pipeline program; saving them from performing the same operation over and over again.

All the interactive tools API methods are designed to be mesh independent. That means the operations are not defined in terms of the mesh geometric elements but in terms of the current point of view in the 3D viewer and the mouse movements performed by the user.

As an example, the lasso select tool could have been implemented as a viewer only operation. Knowing the current camera parameters and the path traced by the user, we could check for all the elements that lie inside the lasso area and call `select_elements` on an empty selection. That would mean that any change on the current mesh would completely invalidate this operation: the API call would no longer select all the elements in the lasso area because some elements may have moved or the indices could be completely different between to different meshes.

Instead, we added the `lasso_select_vertices` method to the *Mesh* class (see section 3.1.2), along with similar methods for all the element types. These methods only receive as parameters the camera settings (position, rotation, FOV, etc.) at the moment of the operation, and a set of screen-space points that form the lasso selection path. With this information we can reproduce the original operation independently of any topology changes or even on a completely different mesh.

This property makes *Meshpipe* very suitable for the previously stated workflow: start with small and simple meshes and keep adding complexity until the whole input data is used. This helps the developer focus on specific problematic areas, or

simply speeds up the development process by having lower processing times until a good pipeline is developed and is ready to be tested on the real data set.

Python API bindings

Another way to reduce iteration times is to completely skip the compilation of the pipeline. For this reason, we decided to implement the Python API bindings. Python is an interpreted language, which means there is no need to compile the program to a language native to the CPU: there is an interpreter layer that understands and executes the program straight from the program's source code.

This interpreter adds some performance overhead, but we think that the benefit of skipping compilation compensates the performance cost. Especially when working with smaller meshes (as in the intended workflow) since the processing time is so short the performance deterioration is almost negligible.

Result inspection

So far we proposed solutions for two steps of the typical testing iteration cycle. The Python bindings get rid of the compilation time, the mesh independent operators allow for working with smaller data sets therefore reducing processing time, so only one step is left to tackle: the inspection of the pipeline results.

There are many great tools available that allow for mesh inspection (see section 2.2.1) but all of them require the user to step out of the development environment, perform whatever steps are needed to load the wanted mesh (usually 2 or more clicks), and only then be able to take a look at the pipeline results.

Instead, *Meshpipe* offers an integrated solution. The same application that runs the processing pipeline has a 3D viewport that can display whatever mesh is being processed as well as the process results. A part from the convenience of not leaving the development environment just to take a look at a mesh, this system has two other advantages: it can display intermediate states, such as selections or properties and it does not need to store any mesh files, since the rendered mesh is already stored in RAM.

Intermediate states are very important, they can help understand what is happening internally when developing a processing pipeline. The same way a debugger can help during the development process by allowing the user to see the state of the program during execution, the *Meshpipe* viewer can be used to visualize what elements belong to a specific selection or display the values of a mesh property by using color scales, as well as overlay a given texture over a mesh. All these display operations can be done either during a pipeline execution or from the viewer's console.

This result and intermediate state inspection part of *Meshpipe* is still not extensively developed. Due to the project's time constraints, the focus has been on settling the the core API and its Python bindings. There is still a lot of room for improvement, we need to support displaying multiple meshes at the same time, a breakpoint system that helps the debugging process and user experience improvements such as auto-completion for the console or an integrated script editor.

3.2.2 User interface

As mentioned above, the user interface is still quite basic. It has been developed into a usable state, but it lacks some user experience improvements that will hopefully come in future versions of *Meshpipe* .

Layout

In figure 3.4, you can see a screenshot of the intended way of using the viewer. Having two windows open side by side: one with a text editor and the other one with the viewer itself allows the developer to view and easily modify the code while having quick access to the viewer display and console. We could have implemented an integrated text editor in the *Meshpipe* viewer, but it would have taken the same place in the screen as the external editor and it would most likely be much less usable than any well established code editor. For the moment, the script editing can be done in a separate window,giving users freedom to choose whatever script editor they prefer and keeping the development scope more focused on the mesh processing aspect.

That being said, having an integrated script editor would have some benefits such as the possibility of adding code breakpoints or partial code execution (execute only the selected parts of the script). So it is not completely out of the picture for future *Meshpipe* versions.

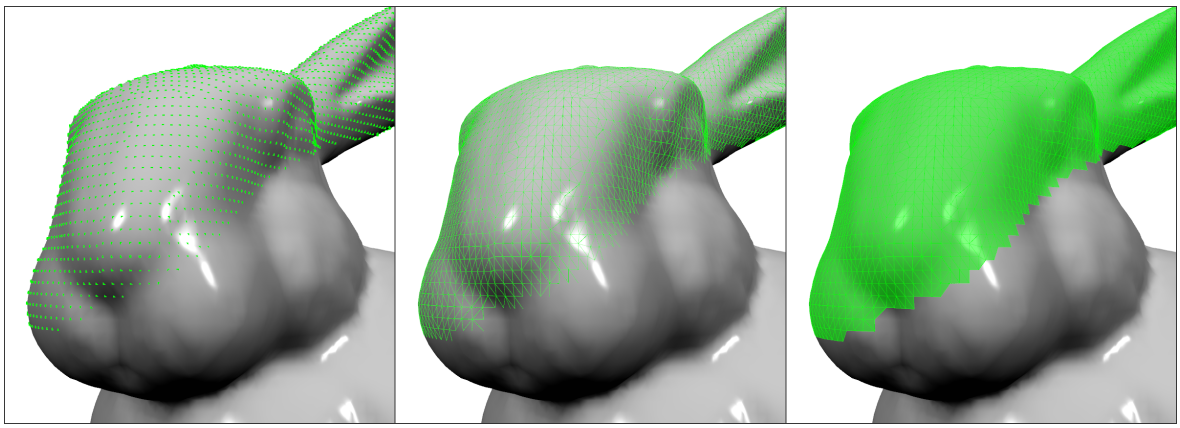

Figure 3.5: Display of different selection types. From left to right: vertex selection, edge selection and face selection.

Viewport

The 3D viewport included in the *Meshpipe* viewer includes all the basic interactions present in many 3D editing software: zoom, rotation and panning. All these interactions can be performed using the mouse or using keyboard shortcuts.

Display-wise, it uses a simple Phong illumination model since we are not seeking realistic or artistic renders but simplicity and clarity of the displayed geometry. It can display wire-frame and point cloud versions of the loaded meshes and also vertex, edge and face selections (see figure 3.5). Finally, it can also display texture data on top of the rendered mesh as well as any of the properties defined on the mesh.

Console

The *Meshpipe* console works the same way as any other python console. It has a global context and allows for executing any python statement, including variable definitions, that then can be used in future statements. The executed lines are printed in green (see figure 3.6) and if the statement returns any value it will be printed in black. Any errors in the input statement or in the executed script will be displayed in the console using a red font. Finally, any mouse operations performed by the user will print their equivalent API call to the console in a dark blue font.

Figure 3.6: Close-up of an example console usage.

This console can be used to test simple operations without having to create a script file. Additionally, it has some extra defined methods that give access to the display capabilities of the viewport. Methods like `display_selection` or `display_texture` change the way the mesh is rendered, adding information in the form of overlays or changing the colors depending on the mesh and texture data.

4. Implementation

In this section we will discuss various topics related to the implementation of *Meshpipe* . Some reasoning behind the implementation decisions as well as a brief explanation on the problems we encountered and how we worked around them. Finally, we will go over the problems that are not solved yet and how we could improve the toolkit even further in future versions.

4.1 Python bindings

One of the key components of the whole *Meshpipe* ecosystem are the Python bindings. We wanted to have an easy to use scripting language for the pipelines but at the same time have a robust and performant implementation of the core data structures and algorithms. For that reason, we decided to use C++ as our core language and Python for pipeline scripting.

C++ is a widely used language in the field of mesh processing, its type strictness and the fact that it compiles directly into native code makes it a great candidate for this type of performance-reliant tasks. In addition to that, there already are many C++ based toolkits for mesh processing development (see section 2.2.2), so we can use one of them as the base for our system and add any missing features or algorithms on top of it.

In our case, we decided to use the *Polygon Mesh Processing* library [9] (section 2.2.2) because of two main reasons: it has a very flexible and performant mesh data structure and it includes a basic 3D viewer that we can build upon. This library serves only as a base and many algorithms and features are implemented by others or by us, for example: the point cloud API or the whole selection system.

In order to expose all the functionality to Python we started evaluating various possible solutions. There are plenty of tools to generate the necessary glue to interface Python and C++, many of which automatically generate bindings from the unedited C++ source. We did not want that, we wanted to be able to control which parts of the API were exposed to Python and which were not. *PMP* includes various basic mesh processing algorithms, but in some cases their implementation does not fully suit our needs. In order to give the user the best possible experience, we added external libraries that overlap with *PMP*'s functionality. Exposing everything to the Python API would mean that some functionality would be duplicated, making the API harder to use.

```
void Mesh::py_def(py::module &m) {
  py::class_<Mesh> mesh(m, "Mesh");
  mesh.def(py::init<>());

  // IO
  mesh.def("read", &Mesh::read, "path", "flags")
      .def("write", &Mesh::write, "path", "flags");

  // Basic selection
  mesh.def("empty_vertex_selection", &Mesh::empty_vertex_selection)
      .def("empty_edge_selection", &Mesh::empty_edge_selection)
      // ...
      .def("select_faces_by_expr", &Mesh::select_faces_by_expr, "expr");

  // Modify selection
  mesh.def("vertex_to_edge_selection", &Mesh::vertex_to_edge_selection,
           py::arg("vertex_sel"), py::arg("both") = false)
      // ...
      .def("vertex_to_face_selection", &Mesh::vertex_to_face_selection,
           py::arg("vertex_sel"), py::arg("min_amount") = 1);
  // ...
}
```

Listing 14: Extract from the *Mesh* class bindings.

For that reason, we decided to take advantage of the *PyBind11* library [12]. It gives us the control we need while, at the same time, being relatively easy to work with. In listing 14 you can find an extract from the bindings in the *Mesh* class.

As you can see, the binding code is extremely simple: just a class declaration and a series of method calls to define what methods will be available in the Python API. On top of that, it also supports the definition of C++ vectors as native Python lists as well as automatic casting between Python and C++ object types.

Although this automatic casting is very convenient it also has a pretty significant performance cost. Every time a variable is converted from a python object to its C++ counterpart or vice-versa the glue code takes a bit of time to do the casting. While this is barely noticeable for a single conversion, it can really slow down the execution if the conversions happen on every iteration of a long loop.

For that reason we have been very careful to minimize the automated castings. By using constant references wherever possible, we can ensure that an object lives entirely on the C++ side or the Python side, but sometimes the conversion is inevitable. In our case all the methods that take a lambda function as an argument are significantly slower than similar methods that can avoid the conversion.

4.2 Viewer implementation

In the planning stage of *Meshpipe* 's development, we considered implementing a simple 3D mesh viewer from scratch using *OpenGL* as the graphics API. However, once we found out *PMP* already offered a boilerplate viewer implementation, we decided to go with it. The mesh processing aspect of *PMP* was similar to other libraries like *OpenMesh*, but the fact that it integrated a simple yet extensible 3D viewer made a really big difference.

Not having to worry about window management, shader compilation and GPU data handling really made it easy for us to focus on the actual features we wanted to implement. The basic *PMP* viewer supports displaying one single mesh with all the typical visualization interactions: panning, rotation and zoom. It also includes different mesh display modes, such as wireframe, point cloud and textured visualization. All we needed to add in the 3D viewport code was the display of the various selection types (see figure 3.5).

One main feature that is not present in the basic *PMP* viewer is the Python console. Luckily for us, all the user interface elements in the viewer are rendered using *IMGUI*, a simple and highly extensible "immediate mode" UI library. This library is widely used in graphics related software because it is OS and rendering API agnos-

tic and very easy to integrate on any project.

Due to its popularity in the game development world, there are open source examples of interactive consoles developed using *IMGUI*. We used one of these examples as a starting point to add our own Python console. The integration was extremely simple and we added all the required functionality to execute Python commands and to display all the application messages. The console includes a command history to quickly re-execute or edit previously issued commands, but it still lacks command auto-completion.

4.3 Problems during development

The development process of *Meshpipe* has been relatively straight-forward. Of course, at the early stages we took some decisions that were quickly deemed wrong and replaced with better solutions, but there were no major setbacks except for one.

When we were implementing the data transfer feature, we needed to cast rays onto meshes and find their intersection points. Instead of implementing and maintaining our own ray intersection algorithm we decided to use an already available and faster alternative. Our first candidate was *Embree* [13], a ray tracing library developed by *Intel*. This library takes advantage of the SIMD instruction set found in modern CPUs to accelerate the traversal of Bounding Volume Hierarchies (BVH [14]). This library includes a lot of low level optimizations that make it perform much better than any custom solution we could have implemented in the duration of this project.

However, once most of the *Embree* integration code was already laid out, we found out that it caused problems with *PyBind11*, the library we use for bindings generation. The C++ integration was working without issues, but any calls to the *Embree* library that came from a Python context resulted in a corrupted memory heap and an application crash.

After weeks of vain debugging efforts, we decided to search for an alternative implementation. We finally chose *NanoRT*, a single file BVH implementation for ray

tracing. It doesn't take full advantage of the CPU hardware as *Embree* did, but it is much more portable and it did not interfere with our Python bindings.

5. Results

Even though *Meshpipe* is not feature complete yet, it is already usable for a large variety of tasks. In this section we will present two use case examples with their respective Python implementations. These examples are inspired by real processing pipelines that have been developed by our research group in the past using other mesh processing tool-kits.

5.1 Automation of a LOD grid

In computer graphics, one very widely used technique to improve performance on large scenes is to have multiple levels of detail (LOD) for each object inside them. This way, we can render the closest objects with the highest amount of detail, while the furthest ones are rendered using the lowest LOD.

This helps balance the resources usage while keeping the overall appearance of the scene: objects further away look smaller and are mostly ignored by the user so they can take the quality degradation without it being noticed.

In our use case we also want to use levels of detail, but instead of a large scene, we want to apply it on a large scanned mesh. We put ourselves in an hypothetical situation where we are developing an interactive museum application. It should allow the user to focus on any part of the art piece (in our case an ancient Greek sculpture), so the amount of detail needs to be kept as high as possible to withstand close-up inspection.

However, rendering the full mesh takes a lot of resources while the user is only focusing on a specific part. This is where *Meshpipe* can be helpful. We want to create a simple pipeline that splits the input mesh into a gird of smaller meshes (see figure

Figure 5.1: Top: original mesh. Bottom: visualization of the 5 simplified cells generated by the pipeline.

5.1) and, for each of them, generates a high and low level of detail version of them.

This would allow our hypothetical application to render only the focused part in full detail, while keeping the context of the rest of the piece by rendering the surrounding parts in low detail. You can find the full pipeline Python code in listing 15.

This pipeline has two configurable parameters, the variables m and n, that define the amount of cell subdivisions in each axis. For our test model we have chosen a grid of $5x1$, since the model is quite elongated. I this pipeline we iterate over each cell computing its bounding box, then select all the faces belonging to it and extract this selection as a new separate mesh. Then, for each extracted mesh, we generate a lower detail version of it using the simplification module.

In order to avoid artifacts where two cells join together, we take advantage of the fact that we can specify which vertices can be collapsed and which can not. We first generate a boundary selection and then invert it, this will ensure that the boundary of the mesh will remain intact while the rest of it can be freely simplified.

Once we have the simplified version of a cell, we use the property transfer module to produce two textures: one color texture and one normals texture. These will be

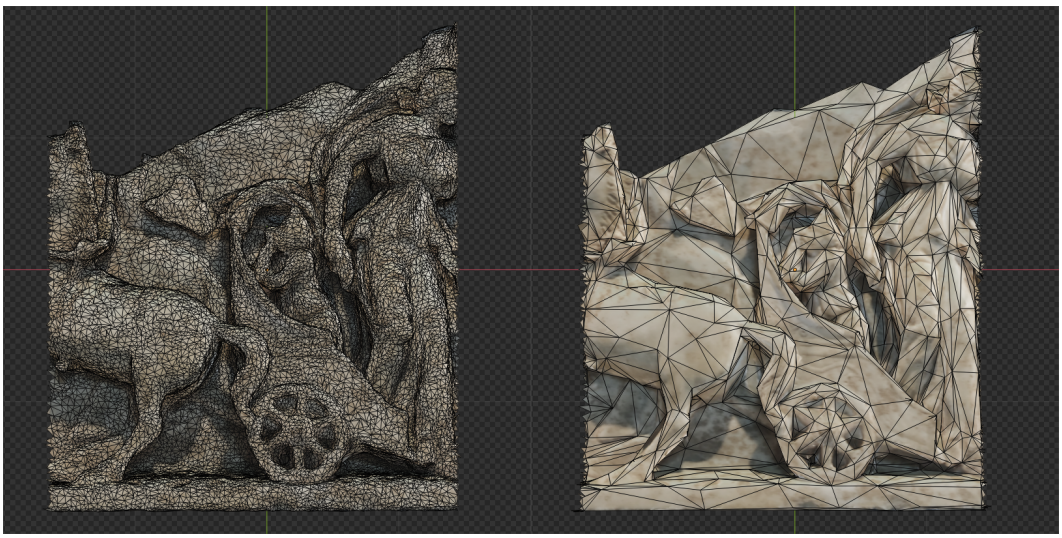

Figure 5.2: Topology detail comparison. Left: original. Right: simplified.

Figure 5.3: Comparison of the original mesh and all the simplified cells together. Top: original. Bottom: simplified cells.

used by the final application to render the low detail meshes with correct colors and more detailed illumination respectively. Finally, all that remains to be done is save the two versions of the cell mesh with a unique name.

With this simple pipeline, we reduced the 125k vertices of the original mesh to almost 6.5k vertices (see figure 5.2) while keeping a lot of the original detail (see figures 5.3 and 5.4) and, most importantly, having a completely automated pipeline. If we wanted to process a different mesh, all we would have to do is change the grid size to an appropriate value and execute the pipeline.

Figure 5.4: Simplification close-up, most of the detail is lost in the silhouettes. Top: original. Bottom: simplified.

```
from meshpipe import MeshPropertyTransfer as mpt
from meshpipe import MeshSimplification as ms

def interpolate_texture_color(bary, he, p):
  uv = Vec2()
  for i in range(3): uv += bary[i] * uvs[he[i]]
  return tex.sample(uv)

def cell_select(v, min_p, max_p):
  p = mesh.position(v)
  return p.x >= min_p.x and p.z >= min_p.y and
         p.x <  max_p.x and p.z <  max_p.y

def process_cell(cell_mesh, i, j):
  simp = cell_mesh.duplicate()

  v = simp.select_boundary_vertices()
  v.invert()
  ms.simplify(simp, v, v.selection_size()*0.05)

  simp.triangulate()
  simp.generate_uvs()

  fid = "{}_{}".format(i,j)
  mpt.transfer_halfedge_vec3_property_to_texture(cell_mesh, simp,
   cell_mesh.halfedge_vec3_property("normal"), "normals"+fid, Vec2(-1,1))
  mpt.transfer_custom_vec3_property_to_texture(cell_mesh, simp,
   interpolate_texture_color, "color"+fid)

  cell_mesh.write("cell_high_" + fid + ".obj")
  simp.write("cell_low_" + fid + ".obj")

tex = Image.open("greek_color.jpg")
n = 5
m = 1

mesh.triangulate()
mesh.compute_halfedge_normals("normal")
uvs = mesh.get_halfedge_vec2_property("h:tex")

bounds_3d = mesh.bounds()
bounds_min = Vec2(bounds_3d[0].x, bounds_3d[0].z)
bounds_max = Vec2(bounds_3d[1].x, bounds_3d[1].z)
bounds_size = bounds_max - bounds_min
cell_size = Vec2(bounds_size.x/n, bounds_size.y/m)

for i in range(n):
  for j in range(m):
    min_p = bounds_min + Vec2(cell_size.x * i, cell_size.y * j)
    max_p = bounds_min + Vec2(cell_size.x * (i+1), cell_size.y * (j+1))

    cell_selection = mesh.select_vertices_by_expr(lambda v:
                       cell_select(v, min_p, max_p))
    cell_selection = mesh.vertex_to_face_selection(cell_selection, 2)
    cell_mesh = mesh.extract(cell_selection)

    process_cell(cell_mesh, i, j)
```

Listing 15: Full pipeline code for generating a grid of 2-LOD meshes.

5.2 Generation of brick mortar

Another practical example of *Meshpipe* 's capabilities is the generation of a mortar mesh that fills in the gaps between a wall of bricks. This example is inspired by a real project that was developed in our research group. During the planning of an architectural project CAD designs are usually simple surfaces that represent the overall shape of the final building. When the project advances to production, more detailed models are created, even to the point of modeling every individual brick.

When visualizing the interior of such a model, the gaps in between the bricks reveal the outside elements making it unpleasant for the user. So the goal of this pipeline is to generate a mesh that covers all the mortar gaps in a bricked model. This can be achieved with a simple processing pipeline that can be found in listing 16.

This algorithm first generates two neighborhoods: the topological neighborhood and the distance neighborhood. The first one tells us, for each vertex, all its topologically adjacent vertices; and the second one all the vertices that lay close to it. With these two sets of neighbors all we do is look for loops of four vertices: if we can reach the same vertex using two different jumps (distance $\rightarrow$ topological and topological $\rightarrow$ distance) then we can create a face joining all the involved vertices.

After that, we do a similar check for distance $\rightarrow$ distance pairs of jumps in order to fill the little square gaps that appear at the intersection of four bricks. The result is a full coverage of the mortar gaps as can be seen in figure 5.5. There are some extensions to the algorithm that can handle staggered brick walls, but they have not been included in order to keep this example short.

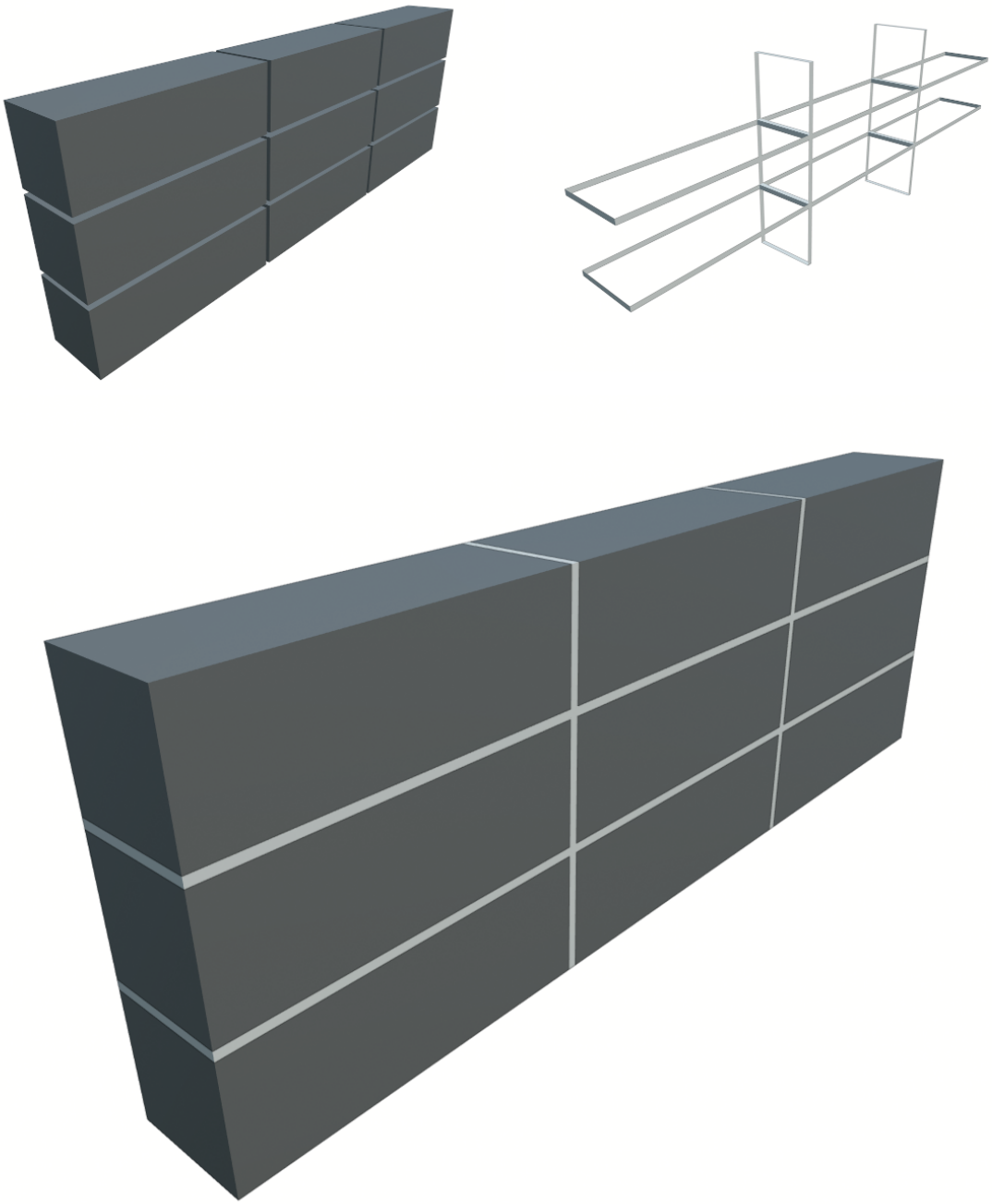

Figure 5.5: Brick mortar generation results. Top left: input mesh. Top right: generated mortar. Bottom: merged result.

```
from meshpipe import MeshPointCloud as mpc

RADIUS = 0.01
seams_mesh = Mesh()
seams_faces = set()

dist_neighborhood = {}
topo_neighborhood = {}

for v in mesh.vertices():
  d = mpc.find_vertices_in_sphere(mesh, mesh.position(v),
            math.sqrt(RADIUS) * 1.05)

  dist_neighborhood[v] = [e[0] for e in d if e[0] != v]
  topo_neighborhood[v] = [vv for vv in mesh.vertices(v) if vv != v]

for v in mesh.vertices():
  dist_topo = set()
  topo_dist = set()
  dist_dist = []

  for dist_vertex in dist_neighborhood[i]:
    topo = topo_neighborhood[dist_vertex]
    for t in topo: dist_topo.add(VertPair(dist_vertex, t))

  for topo_vertex in topo_neighborhood[i]:
    dist = dist_neighborhood[topo_vertex]
    for d in dist: topo_dist.add(VertPair(topo_vertex, d))

  for dt in dist_topo:
    for td in topo_dist:
      if dt == td:
        seams_faces.add(NewFace([v, td.v[0], td.v[1], dt.v[0]]))

  for dist_vertex in dist_neighborhood[i]:
    dist = dist_neighborhood[dist_vertex]
    for d in dist: dist_dist.append(VertPair(dist_vertex, d))

  for dd0 in dist_dist:
    for dd1 in dist_dist:
      if dd0 == dd1:
        seams_faces.add(NewFace([v,  dd1.v[0], dd1.v[1], dd0.v[0]]))

for face in seams_faces:
  new_verts = [seams_mesh.add_vertex(mesh.position(v))
                for v in face.verts]
  seams_mesh.add_face(new_verts)

seams_mesh.write("seams.obj")
```

Listing 16: Full pipeline code for generating a mesh that covers brick mortar gaps.

Conclusions and future work

We started this project with a clear goal in mind: develop a tool that helps the automation of mesh processing tasks. Years of work with polygonal meshes have shown us that there is a real need for automation, but we could not find any existing software or library that focused exactly on that: quick prototyping and pipeline re-usability.

Based on these design goals, we proceeded to develop *Meshpipe* . We took inspiration for all the existing mesh processing tools, but we also wanted to make sure our main goals were met, so we decided to go with a dual approach, having a Python core API for quick prototyping and a 3D viewer that aides the development.

Luckily, we did not face many major setbacks during development, so we were able to integrate quite a lot of features in the library. We have shown that *Meshpipe* is already a tool that can handle real life use cases, but we think there is still room for improvement.

The core API is quite settled, so the clear route for improvement is to add new extension modules. More features and algorithms are always beneficial, and the modules organization ensures that the API does not become bloated or too cumbersome to use.

On the other hand, the 3D viewer is still pretty basic and lacks many of the originally planned features. Hopefully, newer versions of the *Meshpipe* viewer will include support for visualizing multiple meshes at the same time as well as mouse interactive tools that make use of the core API. Finally, the Python console could be improved by adding an automatic completion feature, speeding up even more its usage and making the API more discoverable.

Bibliography

[1] Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In *Eurographics Italian Chapter Conference*. The Eurographics Association, 2008.

[2] Jan Möbius and Leif Kobbelt. Openflipper: An open source geometry processing and rendering framework. In *Curves and Surfaces*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012.

[3] Leif Kobbelt, Stefan Bischoff, Mario Botsch, and Stefan Steinberg. Openmesh: A generic and efficient polygon mesh data structure, 2002.

[4] Sébastien Loriot, Jane Tournois, and Ilker O. Yaz. Polygon mesh processing. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14.1 edition, 2019. URL `https://doc.cgal.org/4.14.1/Manual/packages.html#PkgPolygonMeshProcessing`.

[5] Le-Jeng Andy Shiue. 3D surface subdivision methods. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14.1 edition, 2019. URL `https://doc.cgal.org/4.14.1/Manual/packages.html#PkgSurfaceSubdivisionMethod3`.

[6] Fernando Cacciola. Triangulated surface mesh simplification. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14.1 edition, 2019. URL `https://doc.cgal.org/4.14.1/Manual/packages.html#PkgSurfaceMeshSimplification`.

[7] Sébastien Loriot, Olga Sorkine-Hornung, Yin Xu, and Ilker O. Yaz. Triangulated surface mesh deformation. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14.1 edition, 2019. URL `https://doc.cgal.org/4.14.1/Manual/packages.html#PkgSurfaceMeshDeformation`.

[8] Laurent Saboret, Pierre Alliez, Bruno Lévy, Mael Rouxel-Labbé, and Andreas Fabri. Triangulated surface mesh parameterization. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14.1 edition, 2019. URL `https://doc.cgal.org/4.14.1/Manual/packages.html#PkgSurfaceMeshParameterization`.

[9] Daniel Sieger and Mario Botsch. The polygon mesh processing library, 2019. URL `http://www.pmp-library.org`.

[10] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, (3), July 2002.

[11] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, and M. Tarini. Preserving attribute values on simplified meshes by resampling detail textures. *The Visual Computer*, (10), Dec 1999.

[12] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. URL `https://github.com/pybind/pybind11`.

[13] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, (4), July 2014.

[14] Steven M. Rubin and Turner Whitted. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph.*, (3), July 1980.